Type Systems for Modules
Notes from Meeting #3

Sigurd Schneider
January 13, 2011

Papers


Auxiliary reading:


Summary

Non-Dependent Types for Standard ML Modules

The paper presents second order type theory for the ML module system with generative functors.

The key idea is to use existential quantification over types to account for type generativity to solve the avoidance problem, i.e. avoid leaking local abstract type variables. The solution is to make sure every type containing abstract type variables is prefixed with an existential that binds those abstract type variables. In this way, projections from signature paths can be handled without running into the avoidance problem.

The main result of the paper is to show that the novel type theoretic account is equivalent to the previous state-of-the-art non-standard formulation of the type system. The approach scales naturally to both higher-order and first-class modules.

F-ing Modules

The paper presents a type theoretic account for the ML module system with higher-order generative functors.

It uses the idea from Russo to use existential quantification over types to account for type generativity. The novelty the semantics is defined in terms of an elaboration into $F_\omega$, ultimately showing that ML modules are a particular
mode of use of System F [1]. The notion of paths is generalized to allow module expressions. This is possible, because the type system can explicitly check whether the path can be given a well-formed type that does not mention local abstract types.

The approach offers an understanding of the ML module system in terms of a simple translation into the well studied system $F_{\omega}$. The approach can be extended to allow packing of modules as first-class values, and to incorporate applicative functors.

Questions

1. Q: This week’s papers use elaboration semantics to interpret ML modules in terms of System F types, rather than giving a ”direct” type system in which ML signatures are the types of modules. What are the advantages and disadvantages of this approach?

A: One advantage is that results can be transferred from the target language $F_{\omega}$ to the source language ML modules. In general, this makes sense if either the target language is well understood (and results are in the literature), or the target language is cleaner and allows for easier proofs. A translation can also be easier to understand for people familiar with $F_{\omega}$ (or the target system in general).

A disadvantage is that a translation might not provide insight in how to design the language that is investigated. Direct type systems often give clues to what a language needs. For example, the direct type system approaches to ML modules showed that both, weak and strong sums are not exactly ideal and lead to the development of translucent sums. Only after the nature of the required concept was identified, researchers could embark on understanding this concept in the framework of standard type theory. Another clue that direct type systems offer a lot of insight is the fact that the implementation of the ML module system did things surprisingly “right” wrt. type theory. This might be because the implementation is very close to the language, and provided thus insight in which decisions were reasonable to make.

2. Q: What are the key differences between Russo’s semantics and the one in the F-ing paper?

A:

- In [2] a functor application is an arbitrary structure expression applied to a functor name, whereas [1] assumes that the argument is also a name.

- The binding constructs in Russo’s type language ($\Lambda, \exists, \forall$) take sets of names, in contrast to F-ing which uses syntactically constructed lists. This means that in Russo’s system equality coincides with equivalence. In the F-ing paper, this is not that case, as $\exists\alpha\beta.\Sigma$ is different from $\exists\beta\alpha.\Sigma$. This is not a problem, because equivalence does not come up in the second order semantics, but is replaced by matching, i.e. mutual convertibility: See last rule in Figure 10 in [1], where subtyping is reduced to matching. One obvious advantage of the F-ing
approach is that the type system is an algorithm, whereas Russo’s
type system is an algorithm scheme, which leaves in particular the
realization of the binding constructs to the implementor.

- $F/F_\omega$ only needed in F-ing if required by core calculus, because of
representation of existentials.
- Russo cannot as easily get term translation (because of missing order
on sets, missing order on arguments of functors and hot functions).

3. Q: What’s going on in Section 5.1 of Russo’s paper? Does the issue he
describes here remind you of anything that has been studied recently in
the area of mechanized meta-theory? (This question is directed at those
students who should be familiar with recent work in mechanized meta-
theory – you know who you are.)

A: (discussion was postponed)

4. Q: What is the "avoidance problem" and how do these papers solve it?
   Can you give an example of a program that type-checks in the F-ing Mod-
ules type system but not in the Harper-Lillibridge or Leroy type systems?

A: The avoidance problem is the problem that arises from the fact that
there might be no principal type, (i.e. no most expressive type) of an
module expression that avoids reference to any local abstract type. For
example, if we have $M : \exists \alpha.\{t : \alpha\}$, then we cannot give a principal
type to $M.t$, because any concrete type does not faithfully represent type
abstraction, and any type containing $\alpha$ free is ill formed. On the other
hand, if we have $M : \exists \alpha.\{X : \exists \beta.\Sigma\}$ and $\Sigma$ depends on $\alpha$, then $M.X$
should be given the type $\exists \alpha \beta.\Sigma$. This is exactly the approach taken in [1].

There is a simple syntactic example that does not type-check, because the
other papers do not allow functor applications in paths.

5. Q: In the F-ing paper, if paths $P$ are just arbitrary module expressions
$M$, why bother with the distinction?

A: Because of the avoidance problem. For every path it must be checked
that its signature does not depend on any local abstract type. This is
done by requiring in the premise that there is a type $\Sigma'$ that is equal to
the signature $\Sigma$ (premise $\Sigma \cong \Sigma'$), but does not contain any abstract types
(premise $\Sigma' : \Omega$). If there was no distinction between module expressions
and paths, it would be possible to write down module expressions for
which there is no principal type. (See 4. above)

6. Q: In the F-ing paper, transparent signature ascription and local module
declarations are defined by encodings. Can you work out what the typing
rules for "M:S", "let B in M", and "let B in E" would be if those features
were built directly into the language? (I.e. derive the typing rules for the
encodings and simplify if possible.)
A:

\[
\Gamma \vdash S \sim \exists \pi, \Sigma \quad \Gamma \vdash M : \exists \pi', \Sigma' \sim e \quad \Gamma, \pi' \vdash \Sigma' \leq \exists \pi, \Sigma \uparrow \tau \sim f \\
\Gamma \vdash M : S : \exists \pi', \Sigma' \sim \text{unpack } \langle \pi', y \rangle = e \text{ in pack } \langle \pi', f y \rangle
\]

\[
\Gamma \vdash B : \exists \pi_1, \{l_{X_1} : \Sigma_1\} \sim e_1 \quad \Gamma, \pi_1, X_1 : \Sigma_1 \vdash M : \Xi \sim e_2 \\
\Gamma \vdash \text{let } B \text{ in } M : \exists \pi_1, \Xi \sim \text{unpack } \langle \pi_1, y_1 \rangle = e_1 \text{ in let } X_1 = y_1, l_{X_1} \text{ in pack } \langle \pi_1, e_2 \rangle
\]

7. Q: Explain what "valid", "explicit" and "rooted" signatures are (cf. Section 5.2 of F-ing paper) and why these distinctions are needed.

A: Since higher order unification is undecidable, it is necessary for the type-system to be decidable that no general second order unification problem arises during signature matching (topmost rule on page 5). However, there is a crucial observation that can be exploited to show that the unification problems that arise in the type-system are decidable. The observation is that if all abstract type variables occur in the form of an embedded type field, then every type for which the unification succeeds must have an atomic type signature at the same position.

An abstract type variable \( \alpha \) is rooted if it occurs in the form of an embedded type field \([= \alpha : \kappa_0]\). If all abstract type variables in a signature are rooted, the signature is called explicit. Contra-variance of sub-typing for functors requires that arguments of functor that occur on the left side of the matching must be explicit, too. If this is the case, the signature is called valid.

All these terms are used to show that (i) matching problems only arise for valid LHS and explicit RHS, (ii) in this setting, unification is decidable (Theorem 3+4), namely with the help of the lookup mechanism defined in Figure 15.

References
